Building Integrated ARINC 429 Interfaces using an FPGA

R. Mohan, I. Land

Actel Corporation, Mountain View, California

ABSTRACT

The demand for custom logic within today's avionics industry has led to the increase of Field Programmable Gate Arrays (FPGAs) in aircraft electronic systems. This demand, along with technological advances in size and speed, are resulting in the integration of multiple electronic functions into one chip. ARINC 429, a mature avionics standard and still a very popular communications bus in commercial aircraft, is one such system that is now being integrated with other functions in one chip. To accomplish this, designers are starting to look to pre-built Intellectual Property (IP) to build these System-on-a-Chip (SOC) FPGA designs, also known as Programmable System Chips (PSC).

An ARINC 429 bus interface is used in systems that require a host processor. This paper discusses the integration of a host processor with an ARINC 429 intellectual property core. The bus interface and its host processor functions can both be implemented within one FPGA using pre-built IP to ease integration and shorten the system design process. This paper discusses two examples of ARINC 429 with a host processor. The first example incorporates a high performance 8-bit 8051 microcontroller. The second generalizes that result to an ARINC 429 to Motorola 68000 system.

INTRODUCTION

Bus interfaces such as ARINC 429, MIL-STD-1553, and Ethernet-MAC are used in systems in which there is a host processor controller. The external CPU requirement for the ARINC 429 Bus Interface IP core is necessary to set up the core's control registers and initialize the label memory. Since ARINC 429 operates at either 12.5 KHz or 100 KHz, an 8-bit microprocessor such as an 8051 can fulfill the host requirement.

The following sections will provide a brief overview of ARINC 429 accompanied by background information on the FPGA-based 429 IP core (Actel Core429) and 8051 CPU IP core (Core8051). Similarly, the second design example begins with a description of the Motorola 68000. Both examples start with a high-level system block diagram; discuss interfaces on the ARINC 429 IP and the respective host CPU; and outlines the logic used to connect these blocks together. Along with the hardware considerations, a brief discussion of the CPU to 429 communication protocol is provided. Lastly, a few notes on application software for each respective CPU are included to depict a complete solution.

Figure 1 shows a high-level block diagram of the general system being described. Depending on the specific configuration and host CPU, both the ARINC 429 and the host CPU can reside within one FPGA as shown in Figure 4.

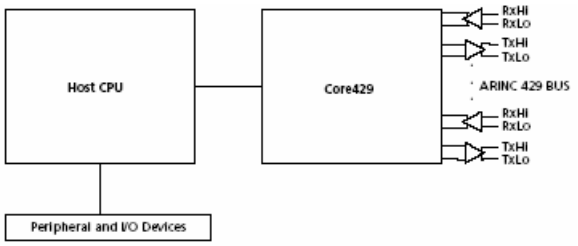


Figure 1 – General 429 to Host Processor System

ARINC 429 Overview

ARINC 429 is a two-wire, point-to-point data bus that is application-specific for commercial and transport aircraft. The connection wires are twisted pairs. Words are 32 bits in length and most messages consist of a single data word. ARINC 429 uses a unidirectional data bus standard (Tx and Rx are on separate ports) known as the Mark33 Digital Information Transfer System (DITS). Messages are transmitted at either 12.5 or 100 kbps to other system elements that are monitoring the bus messages. In addition, ARINC 429 data can be sorted by 8-bit labels to ease communication among different systems.

FPGA Based ARINC 429 IP Core

The 429 Bus Interface IP provides a complete Transmitter (Tx) and Receiver (Rx). The core consists of the three main blocks shown in Figure 2: Transmit, Receive, and CPU Interface. It can be configured to provide both Tx and Rx functions, or either Tx or Rx functions. An FPGA based implementation of ARINC 429 requires external ARINC 429 line drivers and line receivers to interface to the ARINC 429 bus. The use of FPGA based, pre-built IP enables the 429 core to be

highly configurable in terms of the number of Tx and Rx channels used, the clock frequency, the CPU data bus width, and the size of all FIFO's and memories.

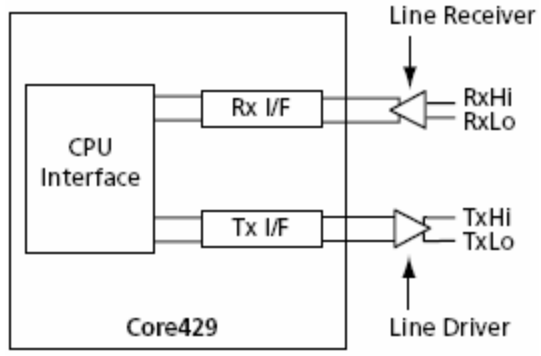


Figure 2 – FPGA Based ARINC 429 IP Core

FPGA Based Host Processor IP Core – 8051

The 8051 host used in the first system example is a high performance, single-chip, 8-bit microcontroller. Figure 3 shows a visual representation of the primary blocks of this 8051 CPU. The core contains internal Special Function Registers (SFRs) which are used to hold various data and control bits. For example, Timer Control, Interrupt Enables, and Serial Port control are some of the uses of the internal SFR memory map. The 8051 IP also contains an SFR Interface which can be used to service up to 101 External SFRs. The External SFRs can be used to interface with an off-core peripheral, such as an ARINC 429 bus interface.

Off-core peripherals use addresses from the SFR address space range 0×80 to 0×FF except those that are already implemented inside the core. When a read instruction occurs with an SFR address that has been implemented both inside and outside the core, the read will return the contents of the internal SFR. When a write instruction occurs with an SFR that has been implemented both inside and outside the core, the value of the external SFR is overwritten. Furthermore, read access to unimplemented addresses will return undefined data, while write access will have no effect.

Note that the SFR address space contains 8-bit addresses, but that the external SFR interface uses a 7-bit bus to transmit the external SFR address (called sfraddr). The MSB of the 8-bit 8051 internal SFR address is not used by the External SFR Interface, instead, it is used to access RAM. When designing the CPU interface to allow the 429 IP to communicate with the 8051 SFR Interface, the external SFRs implemented within the 429 core must be addressed by this 7-bit address. For example, an SFR implemented in the 8051 at address 0×C0 is addressed by the external peripheral at 0×40.

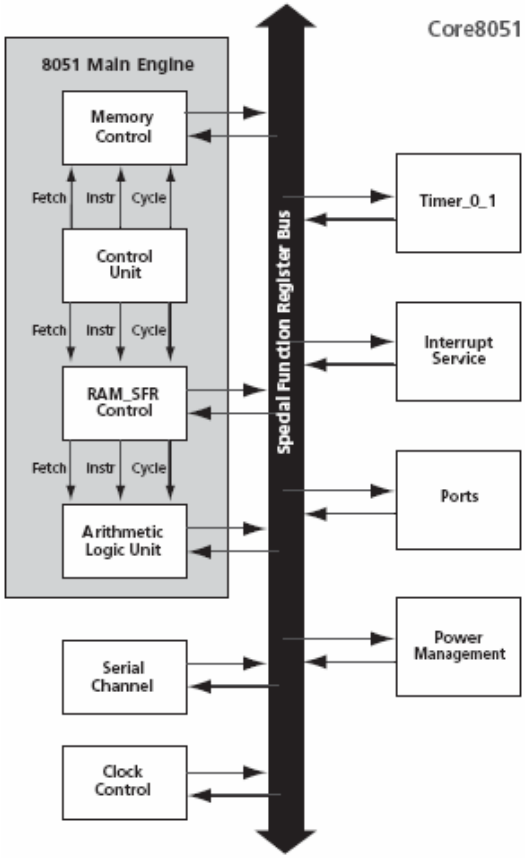


Figure 3 – 8051 Host CPU IP Core

One important consideration is the memory map configuration used by the 8051 CPU. There are four separate memory regions used in the 8051 IP core: DATA, CODE, XDATA, and SFR memory regions. DATA memory is 256 bytes and is used for dynamic storage of program data such as register, stack, and variable data. Although this memory is 256 bytes, typically only the lower 128 bytes are directly addressable for program data. When the upper 128 bytes of DATA memory are addressed, this points to the SFR memory region, which is a combination of internal core memory and external memory for Special Function Registers. CODE space is 64 kb and is used for program storage and interrupt vectors. Lastly, XDATA memory is 64 kb and is used for storage of large data sets, custom-designed peripherals, and extended stack space if necessary. CODE, DATA, and XDATA memory spaces are not part of 8051 IP core and must therefore be implemented by the user, either internal or external to the FPGA.

To overcome the restricted memory map, it was necessary to create an indirect addressing capability. This was done by using memory-mapped registers to hold the address and the corresponding

data. This method has been used in the 429-to-8051 system described below. In fact, four byte-sized registers are used. These registers were conveniently mapped to the 8051 SFR address space.

EXAMPLE 1:
ARINC 429 IP Core Interfaced to 8051 Host

ARINC 429 can be interfaced to an 8051 host by using the external SFR interface mentioned above. One implementation of this ARINC 429 System can be seen in Figure 4. This system requires the use of four SFRs. In order to interface these cores, consideration must be taken when specifying the particular Core429 configuration. For example, the 429 IP used supports CPU data bus widths of 8, 16, and 32 bits. In this system, an 8-bit width is used to match the 8051 bus width. Also, note that the 429 IP core's internal registers are addressed by a 9-bit CPU address which is described in Table 1.

Channel Number				Tx/Rx	Reg Index		Byte Index	
8	7	6	5	4	3	2	1	0
MSB				Bit Positions				LSB

Table 1 – ARINC 429 IP Core 9-Bit CPU Address

Since the SFR Interface has an 8-bit data width, two SFRs are required to hold the 429 9-bit CPU address. In addition, one SFR is required for data written to the 429 Bus Interface by the 8051 and one SFR for data read from the 429 by the 8051. Furthermore, since only one bit is required for the MSB of the 9-bit CPU address, it only occupies bit 0 of the SFR. This leaves bits 1 to 7 available for control/handshaking functions. In this implementation, bit 1 of this SFR is used to indicate whether the requested operation is a read or a write. Bit 2 is used to indicate when the 429 core is busy or done processing the request. The SFRs implemented are summarized in Table 2.

Another advantage of this approach is that a small amount of glue logic can be used to implement any interface logic required if the 8051 and the ARINC 429 cores are in the same clock domain. This system has been physically implemented in the Actel Core429 Development Kit on which both cores operate off a common 16 MHz clock. This is the typical operating frequency of the 8051 core when implemented in an Actel ProASIC^{PLUS} FPGA, and it is also one of the four selectable clock speeds supported by the ARINC 429 Bus Interface IP core.

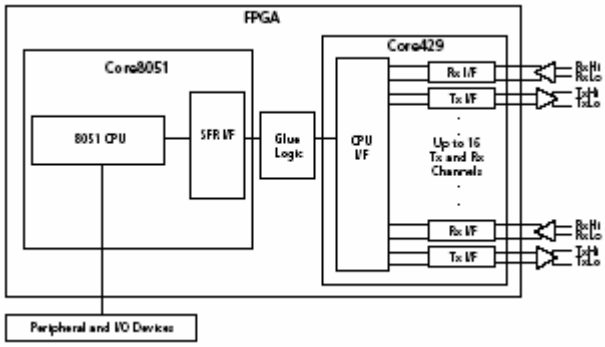


Figure 4 – 429 to 8051 Example System

Core8051 SFR 8-Bit Address	External SFR 7-Bit Address	Function
0xC0	0x40	bits 7:0 of Core429 internal register address
0xC1	0x41	bit 0: MSB of Core429 register address bit 1: 1=Write; 0=Read bit 2: 1=Busy; 0=Done bits 3-7: Not used
0xC2	0x42	byte of data written to Core429
0xC3	0x43	byte of data read from Core429

Table 2 – SFR Usage for the 429 to 8051 System

CPU Interface

The CPU interface allows access to the 429 core's internal registers, FIFO, and internal memory. This allows the system CPU (Core8051) to read/write ARINC data to the FIFO, to access the ARINC 429 control and status registers, and to write to the 429 internal label memory. Table 3 provides a signal description of the Actel ARINC 429 CPU Interface. This interface is synchronous to the 429 core's master clock. Note that CPU data out is asynchronous to the CPU clock for all register reads.

Name	Type	Description
cpu_ren	In	CPU read enable, active low
cpu_wen	In	CPU write enable, active low
cpu_add [8:0]	In	CPU address
cpu_din [i-1:0]	In	CPU data input
cpu_dout [i-1:0]	Out	CPU data output
int_out	Out	Interrupt to CPU, active high
cpu_wait	Out	Indicates that the

		CPU should hold cpu_ren or cpu_wen active while the core completes the read or write operation.

Note: i = CPU_DATA_WIDTH

Table 3 – CPU Interface Signals

Glue Logic

In the ARINC 429 system depicted in Figure 4, the 8051 SFR Interface is tied to the 429 CPU Interface by glue logic. Figure 5 shows a block diagram of this logic.

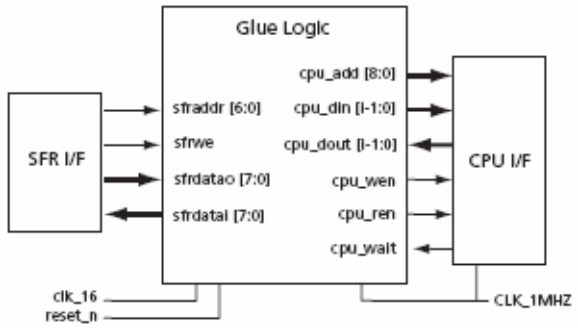


Figure 5 – Core429 to Core8051 Glue Logic

As previously mentioned, the sfraddr is a 7-bit value which is used to address each of the four SFRs implemented. This can be done in verilog by creating parameters to which the sfraddr input can be compared. For example, the following can be used to decode the sfraddr input:

```
// sfraddr for Core429 CPU address [7:0]
parameter [6:0] WRITE_ADDRESS0 = 7'h40;

// sfraddr for Core429 CPU address [8]
// and control bits
parameter [6:0] WRITE_ADDRESS1 = 7'h41;

// sfraddr for data written to Core429
parameter [6:0] WRITE_DATA = 7'h42;

// sfraddr for data read from Core429
parameter [6:0] READ_DATA = 7'43;
```

In addition, any application code running on the 8051 that requires communication with the 429 Bus Interface must follow some sort of communication protocol. For the system shown in Figure 4, the protocol used is as follows:

Writing to ARINC 429 Core from 8051 Host

1. Write 8 bits of the 32-bit ARINC 429 data to WRITE_DATA.
2. Write lower 8 bits of 9-bit Core429 internal address to WRITE_ADDRESS0.
3. Write to WRITE_ADDRESS1 with bit 0 = MSB of 9-bit address, bit 1 = 1 (0 = read; 1 = write).
4. Wait until bit 2 of WRITE_ADDRESS1 = 0 (0 = done; 1 = busy).
5. Repeat steps 1-4 three more times until all 32 bits of ARINC 429 data is written.

Reading from ARINC 429 Core

1. Write lower 8 bits of 9-bit Core429 internal address to WRITE_ADDRESS0.
2. Write to WRITE_ADDRESS1 with bit 0 = MSB of 9-bit address, bit 1 = 0 (0 = read;1 = write).
3. Wait until bit 2 of WRITE_ADDRESS1 = 0 (0 = done; 1 = busy).
4. Read data from READ_DATA.
5. Repeat steps 1-4 three more times until all 32 bits of ARINC 429 data is read.

Note: The 429 core's control and status registers, as well as the label memory, are 8-bit registers and thus only require one read/write per host processor access.

For example verilog code illustrating the hardware implementation of the above communication protocol, see Appendix I and II.

Creating 8051 Application Code to Communicate with the ARINC 429 IP

To successfully create application code that will communicate with Core429, the application code must follow the communication protocol implemented in the hardware. Application code for Core8051 can be written in the C programming language and cross-compiled into 8051 object code by software such as Keil uVision2 and its C51 compiler and 8051 linkers. The software can specifically target the Actel Core8051. In the C-program, the SFRs used in the current implementation are declared as follows:

```
// address for Core429 to decode, bits
// 7:0
sfr ADDR1 = 0xC0;

// bit 0: MSB of 429 addr, bit 1: R/W,
// bit 2: Done/Busy
sfr ADDR2 = 0xC1;

// data written to Core429
sfr DATA_OUT = 0xC2;

// data read from Core429
sfr RDATA = 0xC3;
```

Example C-code functions for ARINC 429 operations can be found in Appendix II.

Verification and Results for ARINC 429 with an 8051 Host

Each core was verified individually on a simulation test bench. Then, the cores were integrated as above and verified against a user test bench and in hardware. In simulation, all tests were passed for each individual core and for the combined entity. After simulation, we connected the blocks, but did not see a handshake between the 8051 and the ARINC 429 IP. A logic analyzer was used to investigate the problem, focusing on the SFR interface because we suspected that was where the problem would reside. After a minor bit of redesign of the glue logic and application code, we were able to achieve a handshake between cores.

Once the handshake was completed, we could then test the setup in a series of tests: from a basic loop back test to tests agains an ARINC 429 standard tester. A Hotek Datair 400 was used to verify operation against the ARINC 429 standard. We have found that standard testers are excellent tools to objectify the development of a core or subsystem. This effort was no exception and our resulting subsystem is documented in the Core429 Development Kit. The kit is used for demonstration, evaluation, prototyping and verification purposes. It contains the application code used in testing, which can be used as is, or as a template. When used as delivered, the 8051 application code included provides the user with an automatic demonstration of ARINC 429 data being transmitted via a 429 bus (using the supplied DB9 cable). This could be used to interface with another ARINC 429 device via the 429 bus and cable provided. The demonstration includes a command mode with a terminal interface that operates via a PC's serial port.

Example 2 – ARINC 429 Interfaced to a Motorola 68000 Host MPU

The following section is a hypothetical example where we discuss integration of a Motorola 68000 device as the host processor.

The M68000 (M68K) microprocessors are available in various configurations. The supported data bus widths include 8 bits (MC68008), 16 bits (MC68000), and 32 bits (MC68020). For the purposes of this discussion, it will be assumed that the M68K used has a 16-bit data bus and a 23-bit address bus. Other features of the M68K include: 16 32-bit Data and Address Registers, 16 MB Direct Addressing Range, 6 Instruction Types, operations on five main Data Types, Memory-Mapped I/O, and 14 Addressing Modes. The five basic data types that can be operated on include bits, binary-coded-decimal (BCD) digits (4 bits), bytes (8 bits), words (16 bits) and long-words (32 bits). Over the various versions of M68K processors, the supported clock speeds include: 8, 10, 12.5, 16.67, 16, and 20 MHz. See Figure 6 for a block diagram of the M68K input and output signals. Note that this diagram shows the M68K address bus ranging from A23-A1. Based on the version of M68K used, this will also be represented as A23-A0 with the restriction that A0 is always driven high in 16-bit mode.

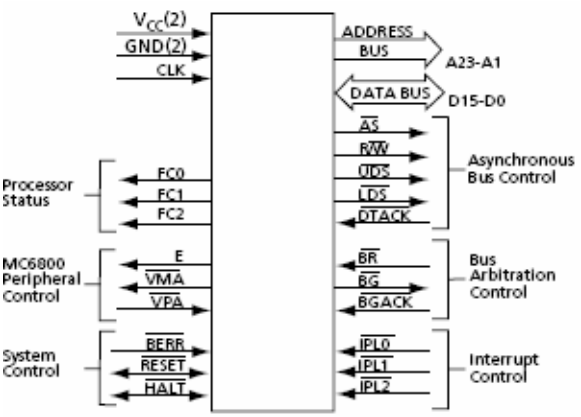


Figure 6 – M68K Input and Output Signals

For a visual representation of the M68K User Programmer's Model, see Figure 7.

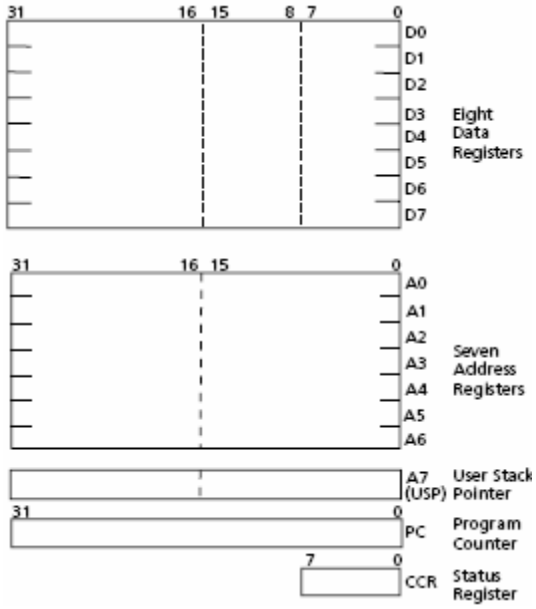


Figure 7 – User Programmer's Model

The registers D0-D7 are used as data registers for byte, word, and long-word operations. The registers A0-A7 are used as address registers that can be used for word and long-word operations. Each address register holds a full 32-bit address. When an address register is used as a source operand, either the low-order word or the entire long-word operand is used, depending on the operation size. When an address register is used as the destination operand, the entire register is affected, regardless of the operation size. If the operation size is word, operands are sign-extended to 32 bits before the operation is performed. Note that A7 is the User Stack Pointer and is used to keep track of the address of the top of the user stack. In the following section, the M68K 16-bit bus operation will be described in more detail. For a more detailed description of the Motorola 68000 family of processors, refer to the *M68000 Microprocessor User's Manual* and the corresponding *Programmer's Reference Manual*.

M68K 16-Bit Bus Operation

Data transfer between devices involves the M68K address bus A1 to A23, the data bus D0 to D15, and the associated asynchronous bus control signals. The M68K address bus is a 23-bit, unidirectional, three-state bus capable of addressing 16 MB of data. The data bus is a 16-bit, bidirectional, three-state bus that provides the general purpose data path for M68K data transfer.

During a read cycle, the processor receives either one or two bytes of data from the peripheral device or from memory. When the instruction specifies a word or a long-word operation, both the upper and lower bytes are read, which requires assertion of both upper and lower data strobes (UDS and LDS respectively). For a byte-sized operation, the M68K uses the internal A0 bit to determine which byte to read. If A0 is zero, the upper data strobe is used, and if A0 is one, the lower data strobe is used. See Figure 8 for a flowchart of the M68K word read cycle.

During a write cycle, the processor sends bytes of data to the peripheral device or to memory. When a word operation is specified, both UDS and LDS are asserted and both bytes are written. For byte operations, when the internal bit A0 is 0, UDS is asserted and when A0 is 1, LDS is asserted. See Figure 9 for a flowchart of the M68K word write cycle.

For more information about M68K bus operation and timing, refer to the *M68000 Microprocessor User's Manual*.

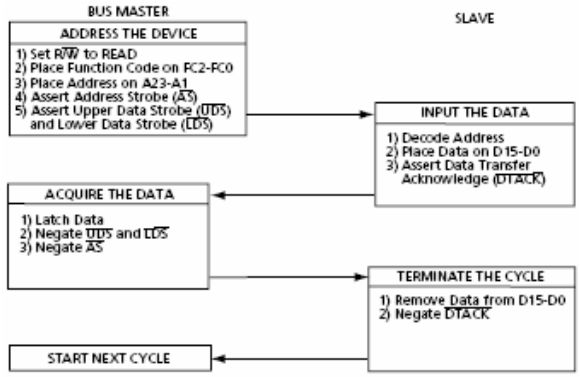

BUS MASTER
SLAVE
ADDRESS THE DEVICE
1) Set R/W to READ
2) Place Function Code on FC2-FC0
3) Place Address on A23-A1
4) Assert Address Strobe (AS)
5) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
INPUT THE DATA
1) Decode Address
2) Place Data on D15-D0
3) Assert Data Transfer Acknowledge (DTACK)
ACQUIRE THE DATA
1) Latch Data
2) Negate UDS and LDS
3) Negate AS
TERMINATE THE CYCLE
1) Remove Data from D15-D0
2) Negate DTACK
START NEXT CYCLE

Figure 8 – 68000 Word Read Cycle Flowchart

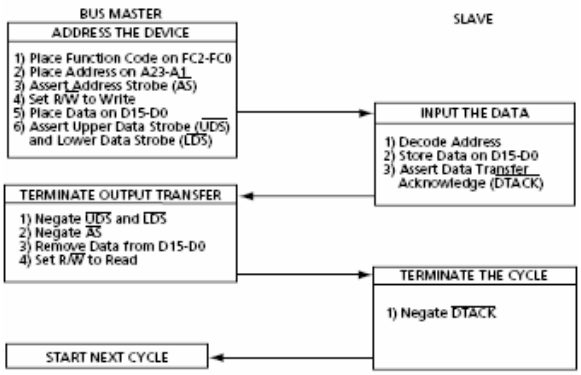

BUS MASTER
SLAVE
ADDRESS THE DEVICE
1) Place Function Code on FC2-FC0
2) Place Address on A23-A1
3) Assert Address Strobe (AS)
4) Set R/W to Write
5) Place Data on D15-D0
6) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
INPUT THE DATA
1) Decode Address
2) Store Data on D15-D0
3) Assert Data Transfer Acknowledge (DTACK)
TERMINATE OUTPUT TRANSFER
1) Negate UDS and LDS
2) Negate AS
3) Remove Data from D15-D0
4) Set R/W to Read
TERMINATE THE CYCLE
1) Negate DTACK
START NEXT CYCLE

Figure 9 – 68000 Word Write Cycle Flowchart

ARINC 429 IP Interfaced to M68K

When interfacing the 429 IP to a M68K host processor, several considerations must taken into account. For example, if using a 20 MHz M68K processor, it would be convenient to configure the 429 IP to also operate at 20 MHz, which is one of its supported operating speeds. Another configuration setting that should not be overlooked is the Data Bus Width. If using an M68K with a 16-bit data bus, the CPU_DATA_WIDTH parameter in the 429 IP should be set to 16 bits. Although there are many configurations possible, interfacing ARINC 429 and the M68K can be accomplished by directly mapping the internal 9-bit CPU address to the relatively large M68K address bus. See Figure 10 for an example 429-M68K system.

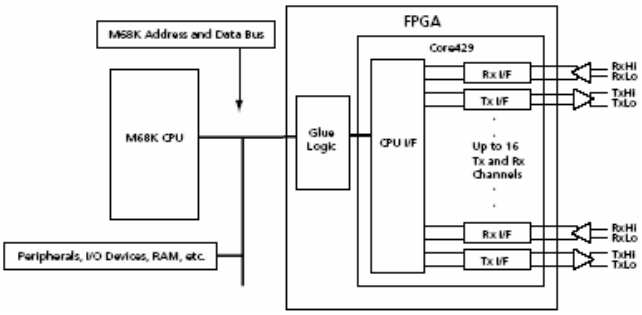


Figure 10 – ARINC 429 IP Interfaced to the M68K Bus (Address and Data Buses)

This interface requires that a 9-bit address space in the M68K memory map be reserved for communication with the 429 Bus Interface IP. For example, one could reserve addresses 0x00000000–0x000001FF for 429 communications unless this address space is pre-reserved by the M68K. Note that if there are less than 8 Tx and 8 Rx ARINC 429 channels implemented, the MSB of the 9-bit 429 address will always be zero. Refer to Table 1 for a more detailed explanation of this 9-bit address.

The approach described above also requires a small amount of glue logic to connect the 429 CPU Interface to the M68K address and data buses.

CPU Interface

The 429 CPU Interface performs the same function as it does when interfacing to the 8051. However, based on the variation of M68K used, the width of the cpu_dout and cpu_din signals will be different. See Table 3 for more information.

Glue Logic

In the 429-M68K system depicted above, the glue logic block is used to interface the 429 CPU Interface to the M68K address and data bus. A block diagram of this logic can be seen in Figure 11. This glue logic maps signals to the protocol described in this section as well as ensures the appropriate control signals are implemented for Core429 timing.

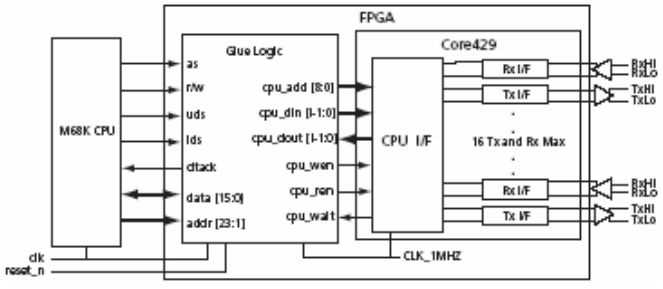


Figure 11 – Core429 to M68K Glue Logic

Notes for Figure 11:

1. *The following signals are active low: dtack, lds, uds, as, & w.*
2. *i = CPU_DATA_WIDTH – 1*
3. *CPU_DATA_WIDTH should match the width of the M68K data (assumed here to be 16 bits).*
4. *In 16-bit mode, the M68K address bus ranges from addr[23:1], otherwise it would be addr[23:0].*

Writing to ARINC 429 IP Core

1. Ensure that addr[23:1] is within the range of addresses reserved for Core429 communication, that the AS input is a logic 0, and that R/W is 0 to indicate a write.
2. If UDS and LDS are logic 0 (along with R/W), then assert cpu_wen and assign addr[9:1] to cpu_add[8:0] and data[15:0] to cpu_din.
3. Wait until cpu_wait is a logic 0.
4. Drive DTACK to a logic 0.
5. When UDS, LDS, and AS are de-asserted and R/W is no longer set to write, de-assert DTACK.

Reading from ARINC 429 IP Core

1. Ensure that R/W is set to read, that addr[23:1] is within the range of addresses reserved for Core429 communication, and that the AS input is a logic 0.
2. If UDS and LDS are logic 0 (with R/W set for read) then assert cpu_ren.
3. Assign addr[9:1] to cpu_add[8:0].
4. Wait until cpu_wait is a logic 0 and pass cpu_dout to data[15:0].
5. Drive DTACK to a logic 0.
6. When UDS, LDS, and AS are de-asserted, stop driving data[15:0] and de-assert DTACK.

Note: The above description is for a 16-bit read/write. For 8-bit operations (on Core429 control registers, status registers, and label memory), only one of either UDS or LDS will be asserted at a time.

Refer to Appendix I for an example of control signals and parameters such as IGNORE_WAIT and cpu_waitm that might be useful considerations when implementing glue logic for a 429-M68K system.

Another important consideration when designing an ARINC 429 to M68K system is the application and use of this system. There are many applications that would be more efficiently implemented by using M68K interrupts to indicate to the program in execution that an ARINC 429 event has occurred. In the 8051 application discussed earlier, the ARINC 429 system was part of a terminal interface that continually waited for a user to issue commands, via a keyboard, and the polling approach

was an acceptable solution. In fact, most applications benefit from the use of interrupts (including with an 8051 controller) over the register polling approach used in Example 1 because it frees up the processor to run other applications. Therefore, to completely specify the glue logic that would interface the ARINC 429 IP to a M68K processor, the interrupt lines IPL0, IPL1, and IPL2 would have to be interfaced with Core429. Refer to the *Motorola M68000 Microprocessor User's Manual* for an explanation of how to implement 68000 interrupts. Refer to the *ARINC 429 Bus Interface* datasheet for information on Core429 interrupt generation.

Creating M68K Application Code to Communicate with the ARINC 429 IP Core

There are several considerations that should be made when creating M68K application code for a 429-M68K system. The software protocol used is based upon the hardware connections made in the glue logic discussed above. Whether or not the M68K assembly code is part of an Interrupt Subroutine (ISR) is determined by the glue logic interface used.

Regardless of the hardware used, the 429 core's internal registers and FIFOs can be more easily used if their hex addresses are mapped to labels such as those described below.

```
* Channel 0 Rx Memory Map
DATA0_RX  equ $00000000 ;Rx Data Register
CNTRL0_RX equ $00000004 ;Rx Control Reg
STAT0_RX  equ $00000008 ;Rx Status Reg
LBL0_RX   equ $0000000C ;Rx Label Memory

* Channel 0 Tx Memory Map
DATA0_TX  equ $00000010 ;Tx Data Register
CNTRL0_TX equ $00000014 ;Tx Control Reg
STAT0_TX  equ $00000018 ;Tx Status Reg
.
.
.
* Channel 15 Rx Memory Map
DATA0_RX  equ $000001E0 ;Rx Data Register
CNTRL0_RX equ $000001E4 ;Rx Control Reg
STAT0_RX  equ $000001E8 ;Rx Status Reg
LBL0_RX   equ $000001EC ;Rx Label Memory

* Channel 15 Tx Memory Map
DATA0_TX  equ $000001F0 ;Tx Data Register
CNTRL0_TX equ $000001F4 ;Tx Control Reg
STAT0_TX  equ $000001F8 ;Tx Status Reg
```

It is useful to keep in mind that the ARINC 429 data width is 32 bits, which requires one long-word operation to read/write. At the same time, operations involving the 429 core's control and status registers, and the label memory require byte-sized operations. However, depending on the version of M68K being used, the data bus size can vary between 8, 16, and 32 bits. For M68K data bus widths less than 32 bits, reads/writes performed on the ARINC 429 Data Registers will have to be repeated until the full 32-bits of data are written. This is controlled by the 9-bit CPU address which includes a map of the byte index for the data being read or written. For a 16-bit data bus, the 429 IP core's data width parameter should be set to 16 and word-sized operations should be used. This would require two successive writes or reads for operations on ARINC 429 32-bit data. This is accomplished by writing to register 0 of the corresponding Tx or Rx channel with the first 16 bits of ARINC 429 data sent to byte index 0 in the 9-bit CPU address. The last 16 bits of data are sent to the address used above, but incremented by 1 to correspond to byte index 1. A similar argument applies for an 8-bit M68K in which 4 read/write byte-sized operations are necessary.

Results for ARINC 429 to M68K

The previous discussion of the Motorola 68000 family of processors and its features, coupled with the 429-M68K system generalization, serve as a starting point for the design and implementation of an ARINC 429 system. The design considerations made for the Actel Core429-Core8051 demonstration design lead into the discussion of interfacing ARINC 429 to a M68K host processor, an unverified effort. The obvious next step would be to build the system and test its operation.

Conclusion

The Actel ARINC 429 IP core (Core429) can be interfaced with other Actel IP, such as Core8051, to create an ARINC 429 system within one FPGA. This can be extended to a generalized 429-M68K system. It also provides a template for how one might integrate other host processors with an ARINC 429 core. Since integration reduces system cost and complexity, it does not have to stop with a 429 to host CPU system - more logic can be added to the FPGA to possibly include a backend application or communication between multiple systems. This is especially important when trying to integrate the fast communication standards of today with pre-existing technology and standards.

Appendix I

Hardware Implementation of Communication Protocol for a 429 to 8051 System:

The following control signals and parameters were used in the glue logic:

```
parameter CPU_DATA_WIDTH = 8;        // Sets the CPU data width
parameter [7:0] PULSE_WIDTH = 1;     // Sets how long the write/read pulse is
parameter IGNORE_WAIT = 0;           // Set to use cpu_wait input from Core429

wire cpu_waitm = cpu_wait && ! IGNORE_WAIT;
```

This protocol can be implemented in hardware as below:

```
always @(posedge clk_16 or negedge reset_n)
  begin
    if (reset_n == 1'b0)
     begin
       cpu_write    <= 1'b0;
       cpu_wen      <= 1'b1;
       cpu_ren      <= 1'b1;
       cpu_din      <= 8'b0;
       cpu_address <= 9'b0;
       datalatch    <= 8'b0;
       cpu_busy     <= 1'b0;
       cpu_start    <= 1'b0;
       cpu_count    <= 0;
     end
   else
     begin
       // Firstly capture the SFR Writes to local registers
       if ((sfraddr == WRITE_DATA) && (sfrwe == 1'b1))
          begin
           cpu_din <= sfrdatao;
          end
       if ((sfraddr == WRITE_ADDRESS0) && (sfrwe == 1'b1))
          begin
           cpu_address[7:0] <= sfrdatao;
          end
       if ((sfraddr == WRITE_ADDRESS1) && (sfrwe == 1'b1) && (cpu_busy == 1'b0) )
          begin
           cpu_address[8] <= sfrdatao[0];
           cpu_write <= sfrdatao[1];          // 1 indicates a write; 0 indicates a read
           cpu_start <= 1'b1;                 // start the cycle
           cpu_busy <= 1'b1;
          end

       // Now do the Access Cycle
       // Must wait for count to get to zero before starting, to make sure ARINC core sees inactive strobe
       if ((cpu_start == 1'b1) && ( cpu_count == 0 ))       // assert cpu_ren or cpu_wen and hold asserted
          begin
           cpu_wen <= ! cpu_write;
           cpu_ren <= cpu_write;
           cpu_count <= PULSE_WIDTH;                  // Miniumn cycle of 31 clocks
           cpu_start <= 1'b0;
```

```
        end
      if ( cpu_count != 0 )
        begin
         cpu_count <= cpu_count -1;
        end
      else
        begin
         if ((cpu_wen == 1'b0 ) && (cpu_waitm == 1'b0 )) // wait for write completes
           begin                                  // cpu_wait is active high
            cpu_wen   <= 1'b1;
            cpu_busy  <= 1'b0;
            cpu_count <= PULSE_WIDTH;                          // delay to ensure inter-write timing
           end
         if ((cpu_ren == 1'b0 ) && (cpu_waitm == 1'b0 ))       // wait for read to complete
           begin
            cpu_ren   <= 1'b1;
            cpu_busy  <= 1'b0;
            datalatch <= cpu_dout;                             // latch the data from the ARINC core
            cpu_count <= PULSE_WIDTH;
           end
        end
    end
  end

// Provide data read-back to the SFR system.
// For test reasons provide full read-back on the register set.
// Note decoding only uses the lowest 2 bits to select from the 4 registers; ensure that this matches the bit
// encodings set above.

always @(sfraddr,cpu_din,cpu_address,cpu_busy,cpu_write,datalatch,int_out)
  begin
    case ( sfraddr[1:0] )
       2'b00  : sfrdatai = cpu_address[7:0];
       2'b01  : sfrdatai = { 4'b0, int_out ,cpu_busy, cpu_write, cpu_address[8] };
       2'b10  : sfrdatai = cpu_din;
       default: sfrdatai = datalatch;
    endcase
  end
```

Appendix II

8051 Application Code for a 429-to-8051 System

In the C-program, the SFRs used in the current implementation are declared as follows:

```
sfr ADDR1 = 0xC0;    // address for Core429 to decode, bits 7:0
sfr ADDR2 = 0xC1;    // bit 0: msb of 429 addr,bit 1: R/W,bit 2: Done/Busy
sfr DATA_OUT = 0xC2; // data written to Core429
sfr RDATA = 0xC3;    // data read from Core429
```

For this particular system implementation and communication protocol, a few low level C Functions can be used to read and write to Core429. Examples of this can be found below:

Note: r_w is a global value declared as: bit r_w;

```
void arinc_addr(short byte, short reg, short tx_rx, uint16 chan)
{
        xdata short addr_pkt1;
        xdata short addr_pkt2;

        if (reg > 0)                    // assemble 9-bit Core429 internal address
        {
                reg = reg * 4;
        }
        if (chan > 0)
        {
                chan = chan * 32;
        }
        tx_rx = tx_rx * 16;

        addr_pkt1 = (chan + tx_rx + reg + byte) & 0x00FF;
        addr_pkt2 = (chan + tx_rx + reg + byte) & 0xFF00;
        addr_pkt2 = addr_pkt2 >> 8;

        if (r_w == 0)           // a read operation
        {
                ADDR1 = addr_pkt1;
                ADDR2 = addr_pkt2;
        }

        else                    //if (r_w == 1)    // write operation
        {
                addr_pkt2 = addr_pkt2 | 0x02;    //set bit 1 = 1 to indicate a write
                ADDR1 = addr_pkt1;
                ADDR2 = addr_pkt2;
        }
}

void arinc_data(long data_ol)           //puts 8-bits of ARINC 429 word into SFR
{
        xdata short data_o = 0;
        data_o = (short) (data_ol & 0x000000FF);
        DATA_OUT = data_o;
}
```

```
void arinc_wait()                    //polls bit 2 of ADDR2 SFR until it is 0, indicating a done state
{
        xdata short rdy = 0;

        rdy = ADDR2;
        rdy = rdy & 0x04;
        while (rdy == 0x04)
        {
                rdy = ADDR2;
                rdy = rdy & 0x04;
        }
}
```

To enforce the communication protocols discussed above, the low level functions arinc_addr, arinc_data, and arinc_wait are used in the manner described below:

Writing from Core8051 Application Code to Core429:

To write 8-bits of data, the following can be used.

```
r_w = 1;
arinc_data(data_out);
arinc_addr(byte, reg, tx_rx, channel);
arinc_wait();
```

To write 32-bits of data, the code above will be written within a loop as below:

```
r_w = 1;
for (byte = 0; byte < 4; byte++)
{
        if (byte == 0)
        {
                arinc_data(data_out);
        }
        else
        {
                data_out = data_out >> 8;
                arinc_data(data_out);
                arinc_addr(byte, reg, tx_rx, channel);
                arinc_wait();
        }
}
```

Reading from Core429:

To read 8-bits of data, the following code can be used:

```
r_w = 0;
arinc_addr(byte, reg, tx_rx, channel);
arinc_wait();
val = RDATA;
```

To read 32-bits of data, the following implementation can be used:

```
r_w = 0;
lim = 4;
for (byte = 0; byte < lim; byte++)
{
        arinc_addr(byte, reg, tx_rx, channel);
        arinc_wait();
        tmp = RDATA;
        data_i = data_i | (tmp & 0xFF);
        if (byte != (lim - 1))
                data_i = data_i << 8;
}
```

Appendix III – Full-size Illustrations

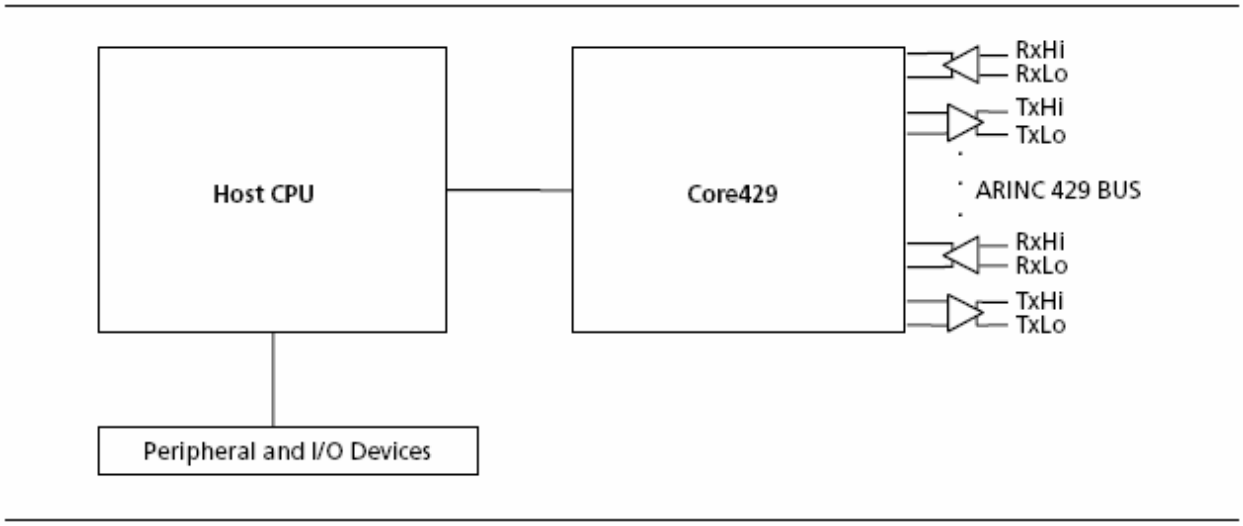


Figure 1 – General Core429 to Host Processor System

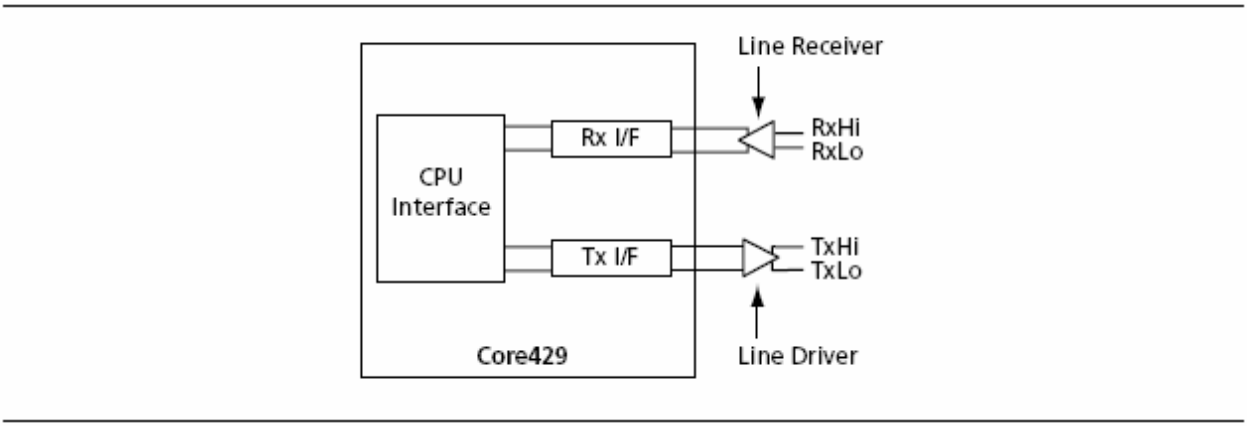


Figure 2 – FPGA Based ARINC 429 IP Core

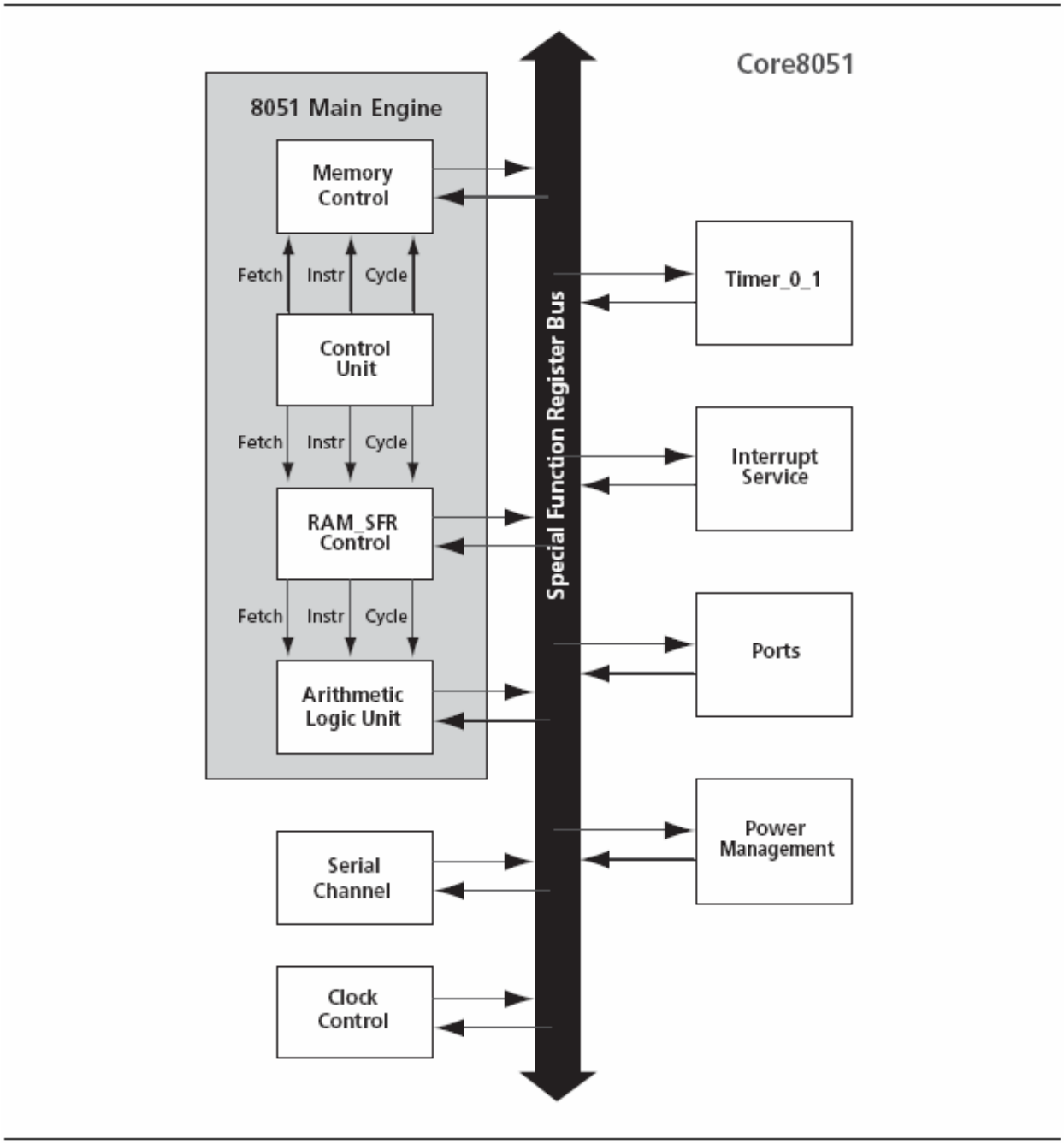


Figure 3 – 8051 Host CPU IP Core Block Diagram

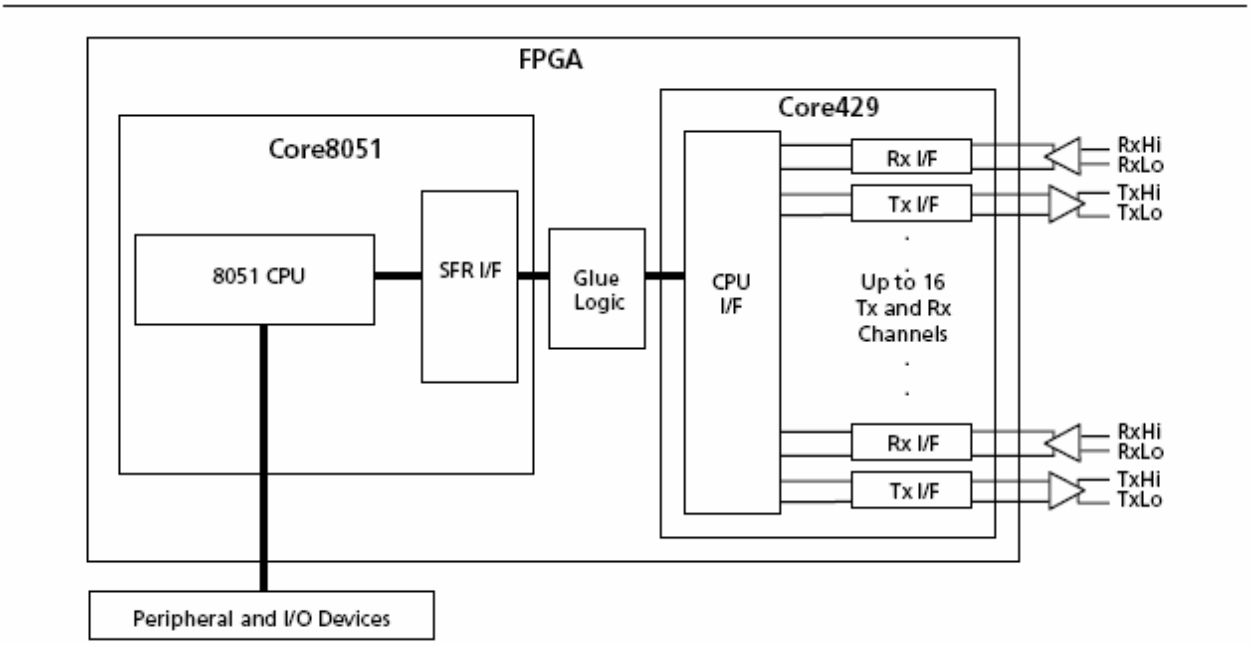


Figure 4 – 429 to 8051 Example System

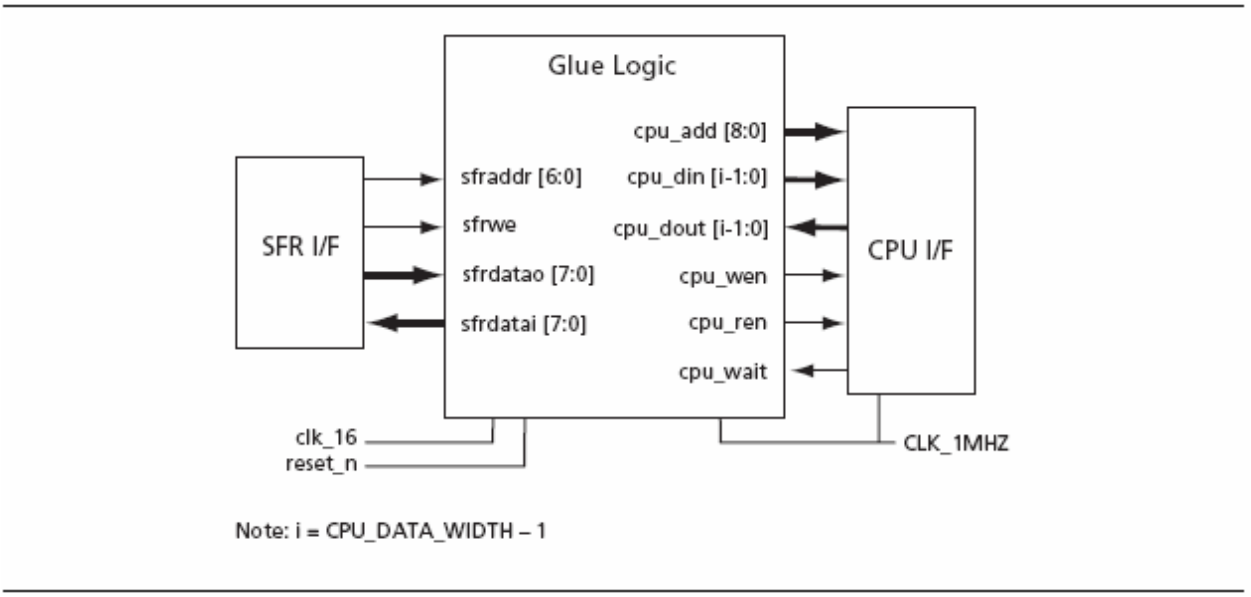


Figure 5 – Core429 to Core8051 Glue Logic

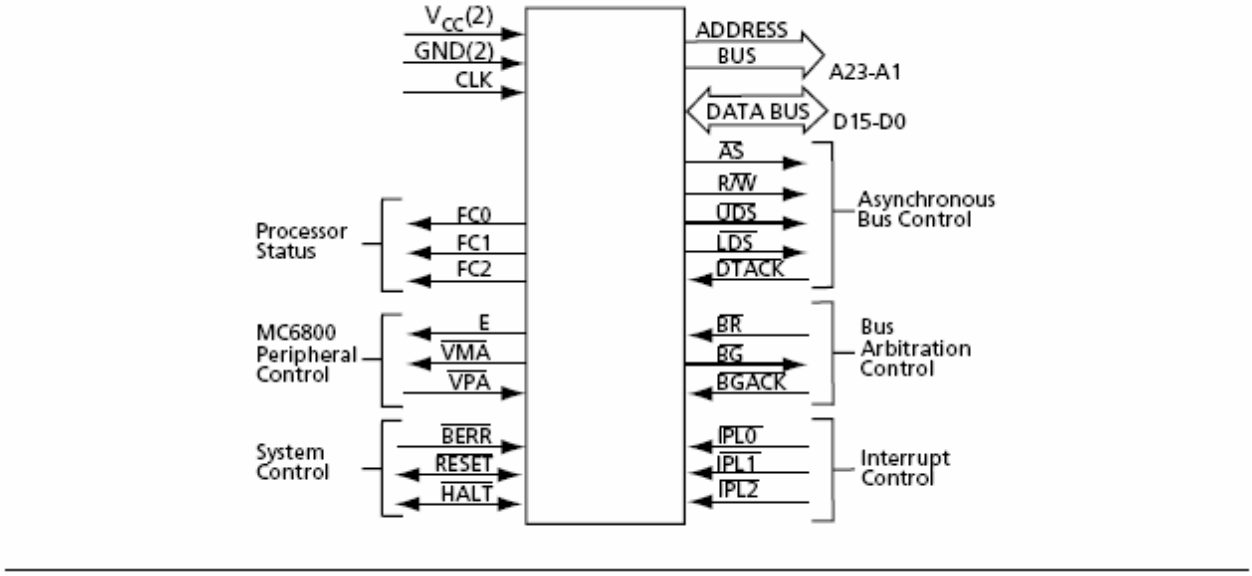


Figure 6 – M68K Input and Output Signals

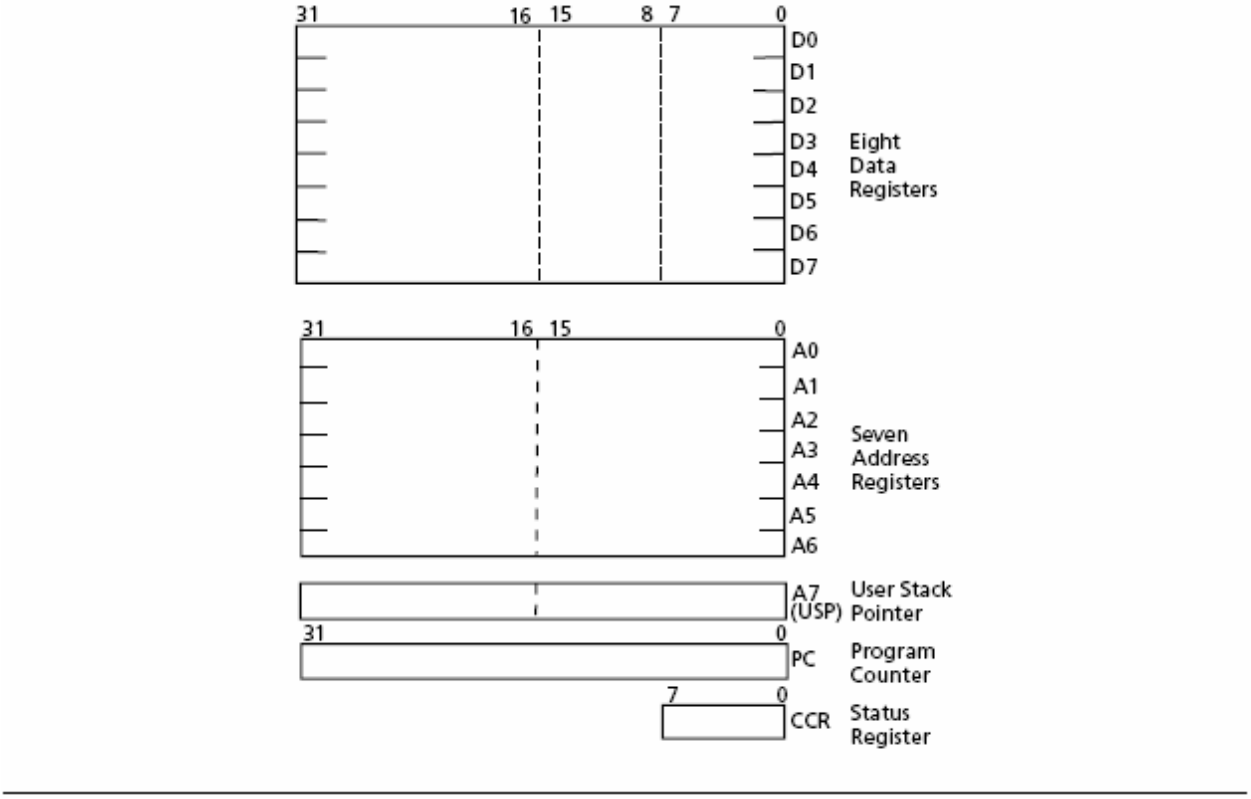


Figure 7 – M68K User Programmer's Model

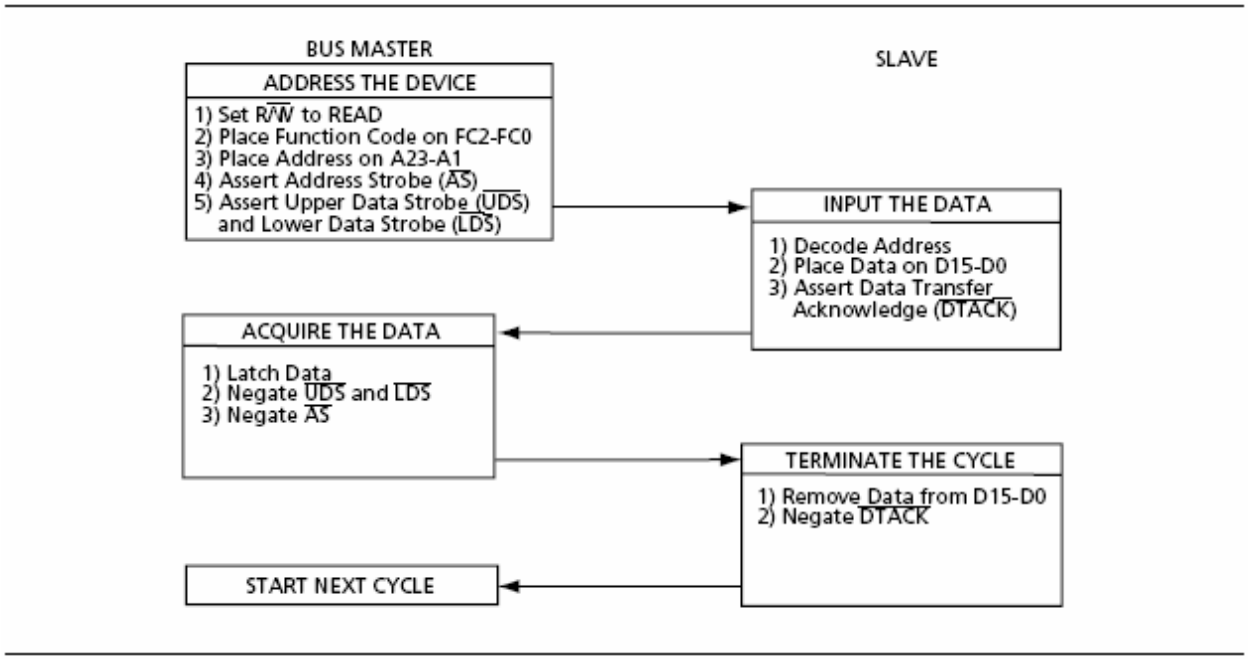

BUS MASTER
SLAVE
ADDRESS THE DEVICE
1) Set R/W to READ
2) Place Function Code on FC2-FC0
3) Place Address on A23-A1
4) Assert Address Strobe (AS)
5) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
INPUT THE DATA
1) Decode Address
2) Place Data on D15-D0
3) Assert Data Transfer Acknowledge (DTACK)
ACQUIRE THE DATA
1) Latch Data
2) Negate UDS and LDS
3) Negate AS
TERMINATE THE CYCLE
1) Remove Data from D15-D0
2) Negate DTACK
START NEXT CYCLE

Figure 8 – 68000 Word Read Cycle Flowchart

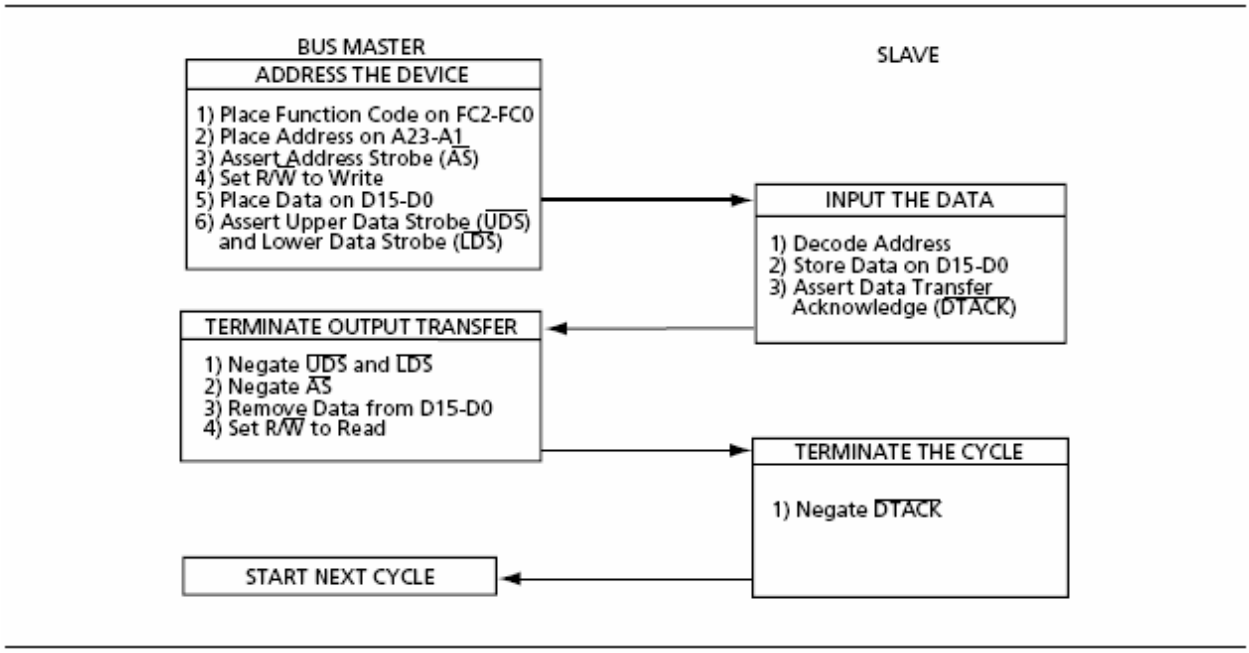
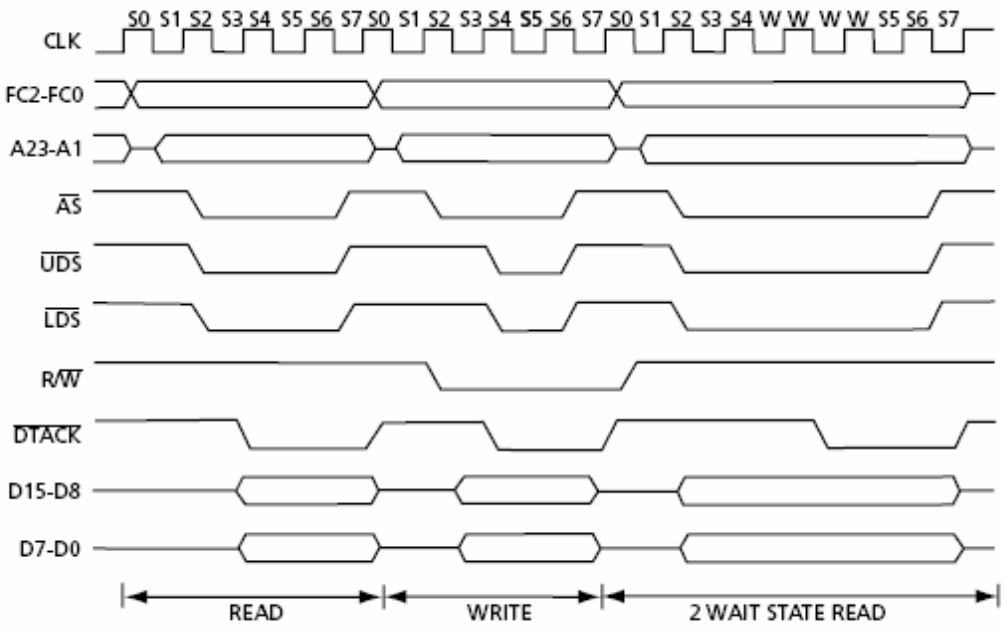

BUS MASTER
SLAVE
ADDRESS THE DEVICE
1) Place Function Code on FC2-FC0
2) Place Address on A23-A1
3) Assert Address Strobe (AS)
4) Set R/W to Write
5) Place Data on D15-D0
6) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
INPUT THE DATA
1) Decode Address
2) Store Data on D15-D0
3) Assert Data Transfer Acknowledge (DTACK)
TERMINATE OUTPUT TRANSFER
1) Negate UDS and LDS
2) Negate AS
3) Remove Data from D15-D0
4) Set R/W to Read
TERMINATE THE CYCLE
1) Negate DTACK
START NEXT CYCLE

Figure 9 – 68000 Word Write Cycle Flowchart



Supplemental Figure – 68000 Read and Write Cycle Timing Diagram

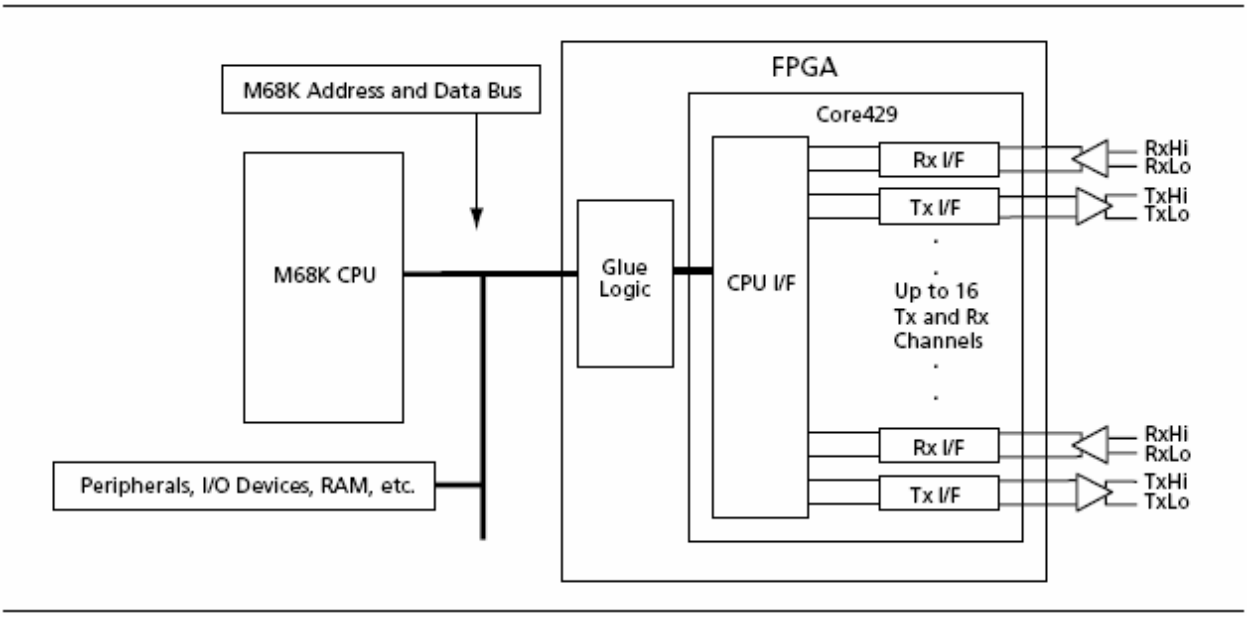
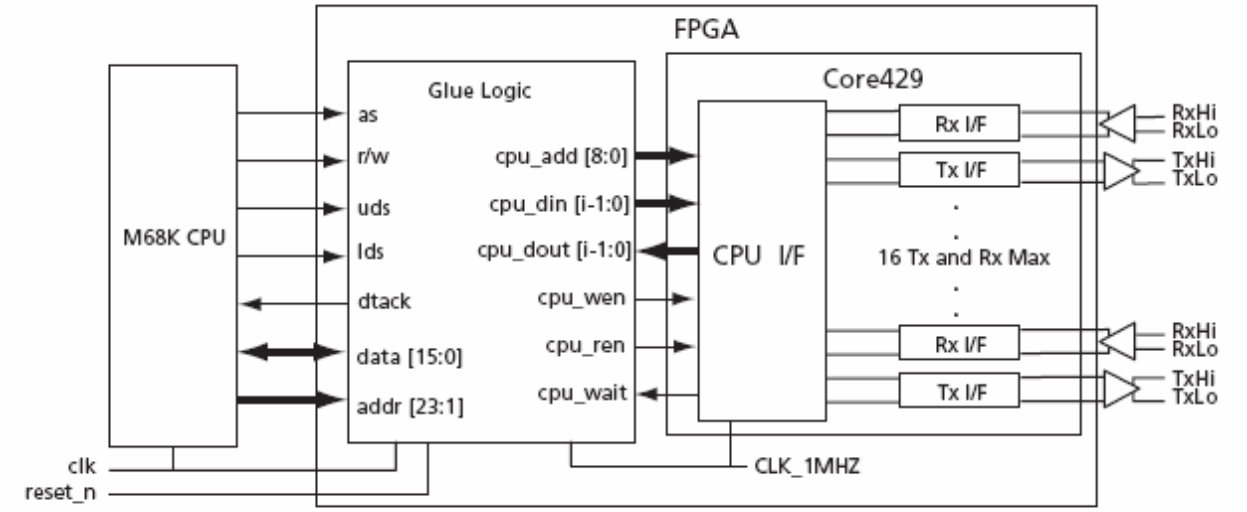


Figure 10 - ARINC 429 IP Interfaced to the M68K Bus (Address and Data Buses)



Notes:

1. *The following signals are active low: dtack, lds, uds, as, and w.*
2. *i = CPU_DATA_WIDTH – 1*
3. *CPU_DATA_WIDTH should match the width of the M68K data (assumed here to be 16 bits).*
4. *In 16-bit mode, the M68K address bus ranges from addr[23:1], otherwise it would be addr[23:0].*

Figure 11 – Core429 to M68K Glue Logic

Sources

Actel Corp., “ARINC 429 Bus Interface Datasheet,” [Online document], 2005 May (v3.1), Available HTTP: http://www.actel.com/ipdocs/CoreARINC429_DS.pdf

Actel Corp., “Core8051 Datasheet,” [Online document], 2005 January (v5.0), Available HTTP: http://www.actel.com/ipdocs/Core8051_DS.pdf

Motorola Inc., “M68000 8-/16-/32-Bit Microprocessors User’s Manual,” [Online document], 1993 (Ninth Edition), Available HTTP: http://www.freescale.com/files/32bit/doc/ref_manual/MC68000UM.pdf